Migrating to Grand Central Dispatch

Session 407

These are confidential sessions—please refrain from streaming, blogging, or taking pictures.

Migrating to Grand Central Dispatch

Kevin Van Vechten
Core OS

Dave Zarzycki
Core OS

Migrating to GCD

- Technology overview
- Single-threaded applications
- Multi-threaded applications
  - GCD and Cocoa
  - Migrating from POSIX

Technology

Overview

Technology Overview

- GCD is part of libSystem.dylib

- Available to all applications

  - `#include <dispatch/dispatch.h>`

Queues

dispatch_queue_t

dispatch_queue_create
dispatch_queue_get_label
dispatch_get_main_queue
dispatch_get_global_queue
dispatch_get_current_queue
dispatch_main
dispatch_apply
dispatch_apply_f
dispatch_async
dispatch_async_f
dispatch_sync
dispatch_sync_f
dispatch_after
dispatch_after_f

Semaphores

dispatch_semaphore_t

dispatch_semaphore_create
dispatch_semaphore_signal
dispatch_semaphore_wait

Objects

dispatch_object_t

dispatch_retain
dispatch_release
dispatch_suspend
dispatch_resume
dispatch_debug
dispatch_get_context
dispatch_set_context
dispatch_set_target_queue
dispatch_set_finalizer_f

Groups

dispatch_group_t

dispatch_group_create
dispatch_group_enter
dispatch_group_leave
dispatch_group_wait
dispatch_group_notify
dispatch_group_notify_f
dispatch_group_async
dispatch_group_async_f

Sources

dispatch_source_t

dispatch_source_create
dispatch_source_cancel
dispatch_source_testcancel
dispatch_source_merge_data
dispatch_source_get_handle
dispatch_source_get_mask
dispatch_source_get_data
dispatch_source_set_timer
dispatch_source_set_event_handler
dispatch_source_set_event_handler_f
dispatch_source_set_cancel_handler
dispatch_source_set_cancel_handler_f

Not Objects

Time

dispatch_time_t

dispatch_time
dispatch_walltime

Once

dispatch_once_t

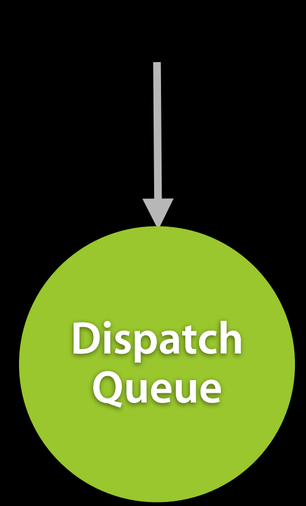
dispatch_once
dispatch_once_f

Queues

- Lightweight list of blocks
- Asynchronous execution
- Enqueue/dequeue is FIFO

Your Thread

A = ^{ ... };



Automatic Thread

Queues

```
dispatch_async(queue, ^{
    /* A */
});
```

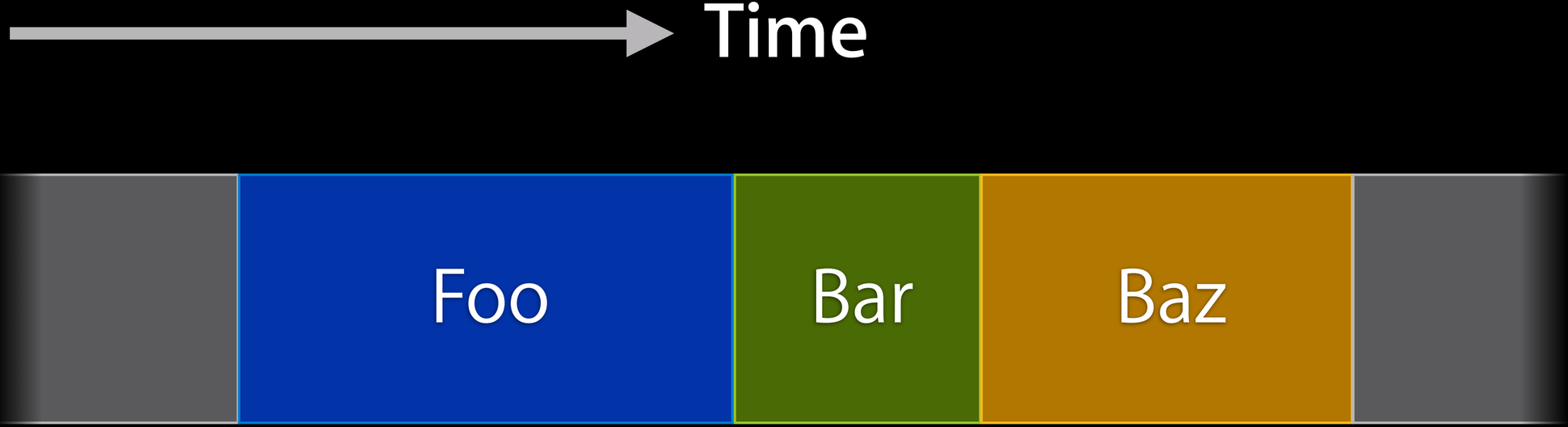
Single-Threaded Applications

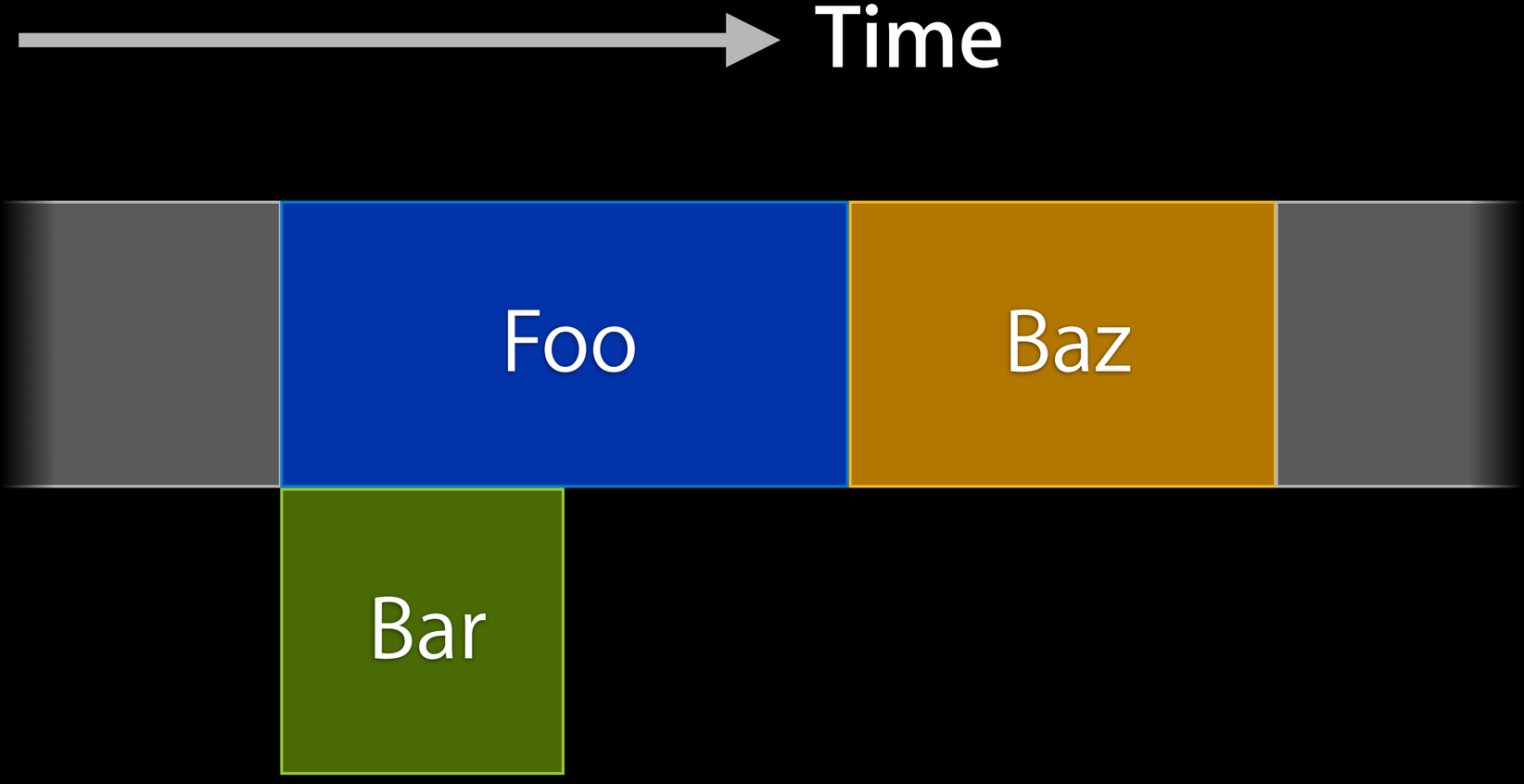
Single-Threaded Applications

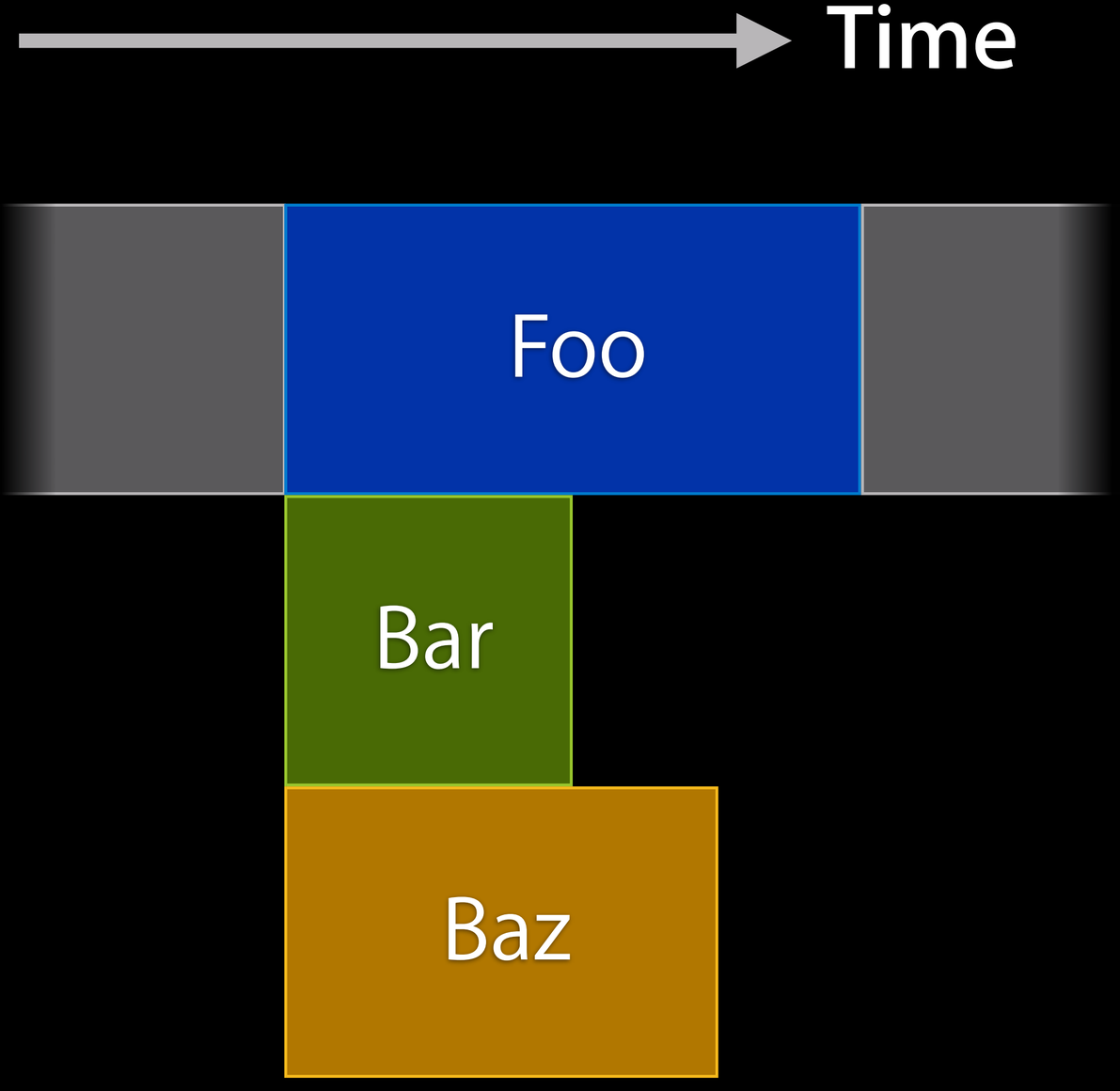
- Identify independent tasks
- Wrap within a block
- dispatch_group_async() the block
- Wait for the group to complete

Single-Threaded Applications

```
result_t r1, r2, r3;

r1 = foo(task1);

r2 = bar(task2);

r3 = baz(task3);

// Use r1, r2 and r3
```


Time
Foo
Bar
Baz


Time
Foo
Baz
Bar


Time
Foo
Bar
Baz

Single-Threaded Applications

```
dispatch_queue_t queue = dispatch_get_global_queue(0,0);
dispatch_group_t group = dispatch_group_create();
__block result_t r1, r2, r3;

dispatch_group_async(group, queue, ^{ r2 = bar(task2); });
dispatch_group_async(group, queue, ^{ r3 = baz(task3); });

r1 = foo(task1);

dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
dispatch_release(group);

// Use r1, r2 and r3
```

```
elapsed = MAX( time(Foo), time(Bar), time(Baz) )
```

Single-Threaded Applications

- Simple application adoption
  - No additional functions
  - No additional data structures
  - No additional heap allocations

Single-Threaded Applications

- Dispatch promises
  - Low overhead
  - Intelligent resource management
- Benefits
  - More CPU cycles available for your code
  - Easy speed gains from concurrency

Multi-Threaded Applications

Multi-Threaded Applications

- Many patterns map to GCD
- Evaluate and adopt on a case-by-case basis
  - No wholesale rewrite required
- Return on investment
  - Productivity
  - Computational efficiency

GCD and Cocoa

GCD and Cocoa

- Running code in the background
  - Avoid the spinning cursor

- Running code on the main thread
  - Synchronize view updates

GCD and Cocoa

Integration

- GCD provides an NSAutoreleasePool
  - No guarantee about how frequently the pool is drained
- GCD coordinates with the garbage collector
  - Dispatch objects are not collectable

GCD and Cocoa

Main thread integration

- The dispatch main queue coordinates with the main run-loop
  - Runs blocks and sources during the common modes
- Dispatch serial queues are non-reentrant
  - Recursively reentering the run-loop from a GCD callback will ***not*** drain the main queue

GCD and Cocoa

Performing selectors

- performSelector:onThread:withObject:waitUntilDone:
- performSelectorOnMainThread:withObject:waitUntilDone:
- performSelectorInBackground:withObject:
- performSelector:withObject:afterDelay:

GCD and Cocoa

performSelector:onThread:withObject:waitUntilDone:

```
// waitUntilDone: NO
dispatch_async(myQueue, ^{
    [myObject doSomething:foo withData:bar ];
});
```

```
// waitUntilDone: YES
dispatch_sync(myQueue, ^{
    [myObject doSomething:foo withData:bar ];
});
```

GCD and Cocoa

performSelectorOnMainThread:withObject:waitUntilDone:

```
// waitUntilDone: NO
dispatch_async(dispatch_get_main_queue(), ^{
    [myObject doSomething:foo withData:bar ];
});
```

```
// waitUntilDone: YES
dispatch_sync(dispatch_get_main_queue(), ^{
    [myObject doSomething:foo withData:bar ];
});
```

GCD and Cocoa

performSelectorInBackground:withObject:

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{
    [myObject doSomething:foo withData:bar ];
});
```

GCD and Cocoa

performSelector:withObject:afterDelay:

```
dispatch_time_t delay;

delay = dispatch_time(DISPATCH_TIME_NOW, 50000 /* 50µs */);

dispatch_after(delay, myQueue, ^{
    [myObject doSomething:foo withData:bar ];
});
```

GCD and Cocoa

- Blocks
  - Call any selector
  - Call multiple selectors
  - No need to pack arguments
  - No need to unpack arguments
- Queues
  - Simple and consistent model for concurrency

Migrating from POSIX

Dave Zarzycki
Core OS

Migrating from POSIX

GCD advantages

- Efficiency
  - More CPU cycles available for your code
- Better metaphors
  - Blocks are easy to use
  - Queues are ***inherently*** producer/consumer
- Systemwide perspective
  - Only the OS can balance unrelated subsystems

Migrating from POSIX

Compatibility

- Mutexes and condition variables are 100% compatible with GCD

- GCD threads are wrapped POSIX threads
  - Do not cancel, exit, kill, join or detach GCD threads

- GCD reuses threads
  - Do not mutate threads without saving and restoring previous state
  - The result of pthread_self() can change between blocks

Migrating from POSIX

In order of complexity

- Once functions
- Threads
  - Fork/join
- Mutexes
- Condition variables
  - Producer/consumer threads
  - Thread pools
- External events
  - select()

Once Functions

Once Functions

- Ensure that multiple threads
  - Execute a function only once per process
  - Wait for the completion of the one function
- Lazy initialization of global variables

Once Functions

```
struct mycontext *global;
pthread_once_t global_once = PTHREAD_ONCE_INIT;
```

```
void init_routine(void) {
        global = malloc(sizeof(struct mycontext));
        initialize_mycontext(global);
}
```

...

```
// ensure global is initialized, but only once
pthread_once(&global_once, init_routine);
```

```
do_something(global->data);
```

Once Functions

```
struct mycontext *global;
dispatch_once_t global_once;
```

...

```
// ensure global is initialized
dispatch_once(&global_once, ^{
        global = malloc(sizeof(struct mycontext));
        initialize_mycontext(global);
});
```

```
do_something(global->data);
```

Once Function

- Convenient
- Efficient

Threads

Threads

- Pthreads execute a function concurrently
  - Avoid blocking the current thread
  - Achieve greater computational bandwidth

Threads

```
pthread_t pthread;
int r;
```

```
r = pthread_create(&pthread, NULL, workfunc, context);
assert(r == 0);
```

```
r = pthread_detach(pthread);
assert(r == 0);
```

Threads

```
dispatch_queue_t queue = dispatch_queue_get_global(0, 0);
```

```
dispatch_async_f(queue, context, workfunc);
```

Threads

- Convenient
- Efficient
  - Thread recycling
  - Deferred based on availability

Fork/Join

Fork/Join

- Background operations
  - Optionally keep the local thread busy
- Waiting for results

Fork/Join

```
pthread_t pthread;
result_t r1, r2;
int r;
```

```
r = pthread_create(&pthread, NULL, background, context);
assert(r == 0);
```

```
r1 = foreground(context);
```

```
r = pthread_join(pthread, &r2);
assert(r == 0);
```

```
// do something with r1 and r2
```

Fork/Join

```
__block result_t r1, r2;
```

```
dispatch_queue_t queue = dispatch_queue_get_global(0, 0);
dispatch_group_t group = dispatch_group_create();
```

```
dispatch_group_async(group, queue, ^{ r2 = background(context); });
```

```
r1 = foreground(context);
```

```
dispatch_group_wait(group, DISPATCH_TIME_FOREVER);
dispatch_release(group);
```

```
// do something with r1 and r2
```

But wait, there's more…

Fork/Join

```
__block result_t r1, r2;
```

```
dispatch_queue_t queue = dispatch_queue_get_global(0, 0);
dispatch_group_t group = dispatch_group_create();
dispatch_group_async(group, queue, ^{ r1 = foreground(context); });
dispatch_group_async(group, queue, ^{ r2 = background(context); });
```

```
dispatch_group_notify(group, dispatch_get_main_queue(), ^{
    // do something with r1 and r2
    dispatch_release(group);
});
```

Threads

- Convenient
- Efficient
- More expressive
  - Join on multiple operations
  - Asynchronous notification

Mutexes

Mutexes

- Enforce mutually exclusive access to critical sections
- Serialize access to shared state between threads
- Ensure data integrity

Mutexes

```
r = pthread_mutex_lock(&mutex);
assert(r == 0);
```

```
// critical section
mybuffer->size += strlen(newdata) + 1;
mybuffer->ptr = reallocf(mybuffer->ptr, mybuffer->size);
if (mybuffer->ptr == NULL) {
    mybuffer->size = 0;
    r = pthread_mutex_unlock(&mutex);
    assert(r == 0);
    return;
}
strlcat(mybuffer->ptr, newdata, mybuffer->size);
```

```
r = pthread_mutex_unlock(&mutex);
assert(r == 0);
```

Mutexes

```
dispatch_sync(mySerialQueue, ^{
    // critical section
    mybuffer->size += strlen(newdata) + 1;
    mybuffer->ptr = reallocf(mybuffer->ptr, mybuffer->size);
    if (mybuffer->ptr == NULL) {
        mybuffer->size = 0;
        return;
    }
    strlcat(mybuffer->ptr, newdata, mybuffer->size);
});
```

But wait, there's even more…

Mutexes

```
dispatch_async(mySerialQueue, ^{
    // critical section
    mybuffer->size += strlen(newdata) + 1;
    mybuffer->ptr = reallocf(mybuffer->ptr, mybuffer->size);
    if (mybuffer->ptr == NULL) {
        mybuffer->size = 0;
        return;
    }
    strlcat(mybuffer->ptr, newdata, mybuffer->size);
});
```

Mutexes

- Convenient
- Efficient
- Safe—cannot return without unlocking
- More expressive
  - Deferrable critical sections
  - Wait-free synchronization

Condition Variables

Condition Variables

- Waiting for application-defined events
- Threads sleep until condition variable is signaled
- Common use is producer/consumer
  - Producer signals consumer when work is added

Producer/Consumer

Producer/Consumer

- POSIX requires you to implement your own model
  - A lot of code
  - No convention between frameworks
    - No interoperability

- Dispatch queues are the solution
  - Formalized API for producer/consumer
  - Trivial interoperability between frameworks

Producer/Consumer

```
void *consumer(void *context)
{
    consumer_item_t item;
    consumer_t q = context;
    int r;
    for (;;) {
        r = pthread_mutex_lock(&q->mtx);
        assert(r == 0);
        while (!(item = TAILQ_FIRST(&q->items))) {
            r = pthread_cond_wait(&q->cond, &q->mtx);
            assert(r == 0);
        }
        TAILQ_REMOVE(&q->items, item, linkage);
        r = pthread_mutex_unlock(&q->mtx);
        assert(r == 0);
        item->func(item->ctxt);
        free(item);
        consumer_release(q);
    }
    return NULL;
}
```

Producer/Consumer

```
void producer(consumer_t q, void *context, void (*function)(void *))
{
    consumer_item_t item;
    int r;
    assert(q);
    item = calloc(1, sizeof(struct consumer_item_s));
    assert(item);
    item->ctxt = context;
    item->func = function;
    consumer_retain(q);
    r = pthread_mutex_lock(&q->mtx);
    assert(r == 0);
    if (TAILQ_EMPTY(&q->items)) {
        r = pthread_cond_signal(&q->cond);
        assert(r == 0);
    }
    TAILQ_INSERT_TAIL(&q->items, item, linkage);
    r = pthread_mutex_unlock(&q->mtx);
    assert(r == 0);
}
```

Producer/Consumer

- A lot of code
- Did ***not*** implement retain/release
- Did ***not*** show data structure definitions
- Did ***not*** show thread creation/cleanup

Producer/Consumer

```
dispatch_async(queue, ^{ /* work */ });
```

Producer/Consumer

- Convenient
- Efficient
- Better use of your time and intellectual energy
- Full integration with OS frameworks

Thread Pools

Thread Pools

- Set of consumer threads
  - Impractical to get the count "right"
- Shares all of the problems with producer/consumer threads
  - Even more code
- Really just a dispatch global queue

External Events

External Events

Loose convention

- Frameworks export a descriptor and a function

- Your code must implement a descriptor based event loop
  - select()
  - kqueue()

External Events

- POSIX condition variables ***cannot*** monitor external events
- Application-defined events must be simulated with external events
  - pipe()
  - socketpair()

External Events

```
for (;;) {
    FD_ZERO(&readfds);
    nfds = 0;
    for (i = 0; i < event_count; ++i) {
        FD_SET(&readfds, events[i]->fd);
        nfds = MAX(i + 1, nfds);
    }
```

```
    do {
        res = select(nfds, &readfds, NULL, NULL, NULL);
    } while (res == -1);
```

```
    for (i = 0; i < res; ++i) {
        if (FD_ISSET(&readfds, i)) {
            events[i]->handle_descriptor(events[i]->context);
        }
    }
}
```

External Events

What GCD provides

- Standardized event tracking
  - Both single-threaded and multi-threaded applications
- External and application-defined events can target the same queue
  - No bridging between frameworks by the application
- No looping or scheduling in application code
  - Dispatch queues manage all of the details

External Events

Single-threaded application

```
int main(void)
{
    dispatch_source_t fdSource;

    fdSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
                  STDIN_FILENO, 0, dispatch_get_main_queue());

    dispatch_source_set_event_handler_f(fdSource, myReadFunc);
    dispatch_resume(fdSource);

    library_init_with_queue(dispatch_get_main_queue());

    dispatch_main();
}
```

External Events

Multi-threaded application

```
int main(void)
{
    dispatch_queue_t background;
    dispatch_source_t fdSource;

    background = dispatch_queue_create(“com.example.background”, NULL);

    fdSource = dispatch_source_create(DISPATCH_SOURCE_TYPE_READ,
            STDIN_FILENO, 0, background);

    dispatch_source_set_event_handler_f(fdSource, myReadFunc);
    dispatch_resume(fdSource);

    library_init_with_queue(dispatch_get_main_queue());

    dispatch_main();
}
```

External Events

Standardized event handling

- Consistent handling of application-defined and external events
  - dispatch_async()
  - dispatch_source_create()
- Supports all kqueue types
  - Descriptors, timers, signals, processes, …
- No application looping and glue code

Migrating to GCD

- Your code is ready to adopt GCD
  - Convenient
  - Efficient
- Adoption à la carte
  - Start small
  - Look for big wins
  - Verify that your goal is accomplished

More Information

Michael Jurewitz
Developer Tools and Performance Evangelist
jurewitz@apple.com

Developer Forums
> Mac Development > Grand Central Dispatch
http://devforums.apple.com/community/mac/gcd

Documentation
Concurrency Programming Guide
http://developer.apple.com/

Related Sessions

Understanding Grand Central Dispatch in Depth	Presidio Thursday 9:00AM
Migrating Your Application to Grand Central Dispatch	Pacific Heights Thursday 2:00PM

Labs

Grand Central Dispatch Lab	Snow Leopard Lab Thursday 3:30PM

Q&A

WWDC Session Survey

You can rate your sessions on the WWDC Attendee Site.

Survey forms are located on each session description page at:

developer.apple.com/wwdc/attendee

WWDC09